Atelier Calcul Intensif

Pôle AM2I

Jérémie Gaidamour (IECL)
Ingénieur de Recherche CNRS
jeremie.gaidamour@univ-lorraine.fr

Décembre 2024

Objectifs

Présentation de techniques et d'outils utiles pour le calcul intensif :

- Comprendre les problématiques de passage à l'échelle
- Présenter quelques solutions pour exploiter le parallélisme *(et souligner l'importance de la gestion mémoire)*
- Faire un tour d'horizon des outils et bibliothèques existantes (*pour s'appuyer sur l'existant*)

Plan

Comprendre les enjeux

Programmation parallèle

Bibliothèques logiciels

Architecture des calculateurs

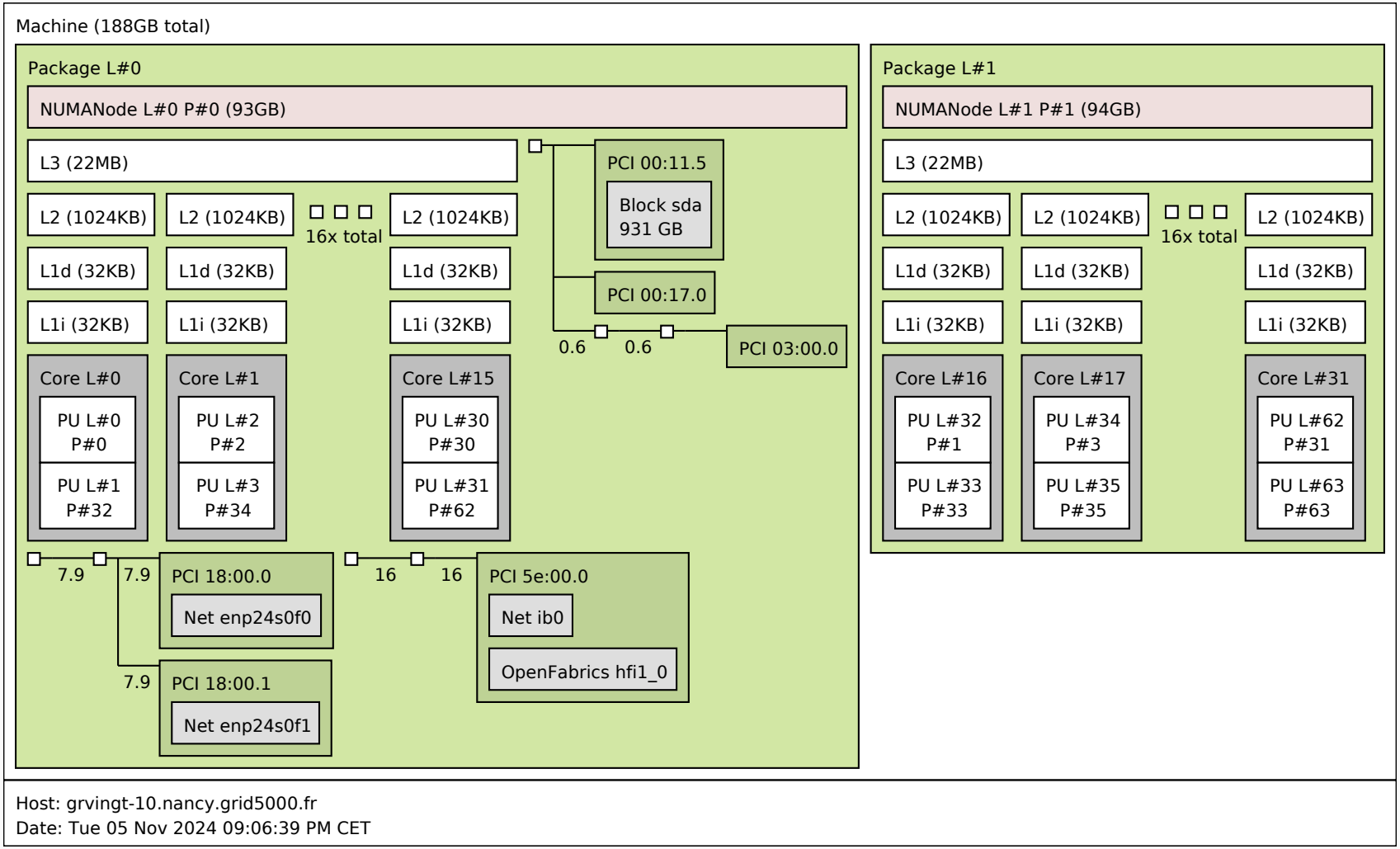
Les machines ne deviennent pas plus rapide !

Enjeu : l'augmentation de la puissance du materiel est liée à la duplication d'unités de calculs *qu'il faut alimenter en données*.

- CPU : fréquence adaptative, pipelines, superscalaire, vectorisation (SIMD), multithreading (SMT)
- Mémoire : caches hiérarchiques, architecture NUMA, accélérateurs, mémoire distribuée...

Architecture des calculateurs

Noeud de 2 CPU, 16 coeurs/CPU, 100 Gbps Omni-Path :



hwloc (lstopo)

Plan

Comprendre les enjeux

Programmation parallèle

Bibliothèques logiciels

Paradigmes de programmation

Typologie des différents types de parallélisme :

- **Instruction Level Parallelism** : exécution concurrentes d'instructions indépendantes au niveau hardware
- Exploitation des unités **SIMD** (vectorisation)
- En mémoire **partagée** : parallélisation de type threads légers
- En mémoire **distribuée** : communication inter-noeuds par échange de messages
- **Accélérateurs** : parallélisation de type SIMT

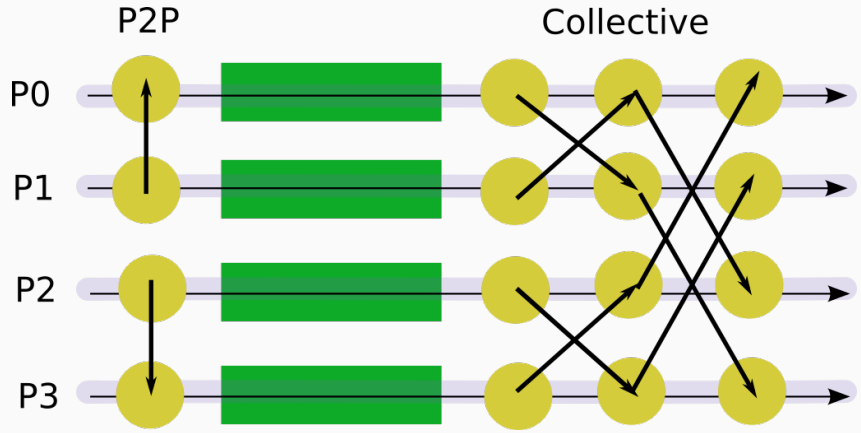
Développement logiciel ?

Idéalement, on cherche une solution portable, performante, simple (coût de dev. faible), stable et pérenne.

En mémoire distribuée

MPI :

- Processus indépendants qui communiquent entre eux.
- Permet de distribuer de larges problèmes (ex : on assigne à chaque processus une partie du problème).
- Messages de communication explicites entre les processus. Les machines distribuées utilisent des réseaux à faibles latences.

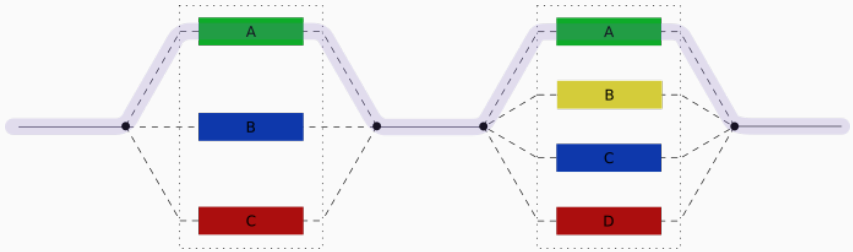


En mémoire partagée

Les **threads** ont accès à une mémoire partagée : ne nécessite pas de transfert de données entre threads, mais nécessite des synchronisations.

OpenMP < 3.0 : programmation SIMD, modèle *Fork-Join* avec des *parallel regions*) :

- permet l'annotation d'un code séquentiel : moyen simple d'exploiter du parallélisme sur un code existant
- gestion explicite des problèmes de concurrences




```
#pragma omp parallel for
for (i=0; i<N; i++) {
...
#pragma omp critical
  globalvar += ...
}
```

img : wikipedia

Exemple de problématiques

En mémoire distribuée :

- Ex. de difficultés : distribution des données, scalabilité liée au ratio surface/volume d'une décompositon, réplications de données, nombre de messages, load balancing...
- *Il est possible d'utiliser un modèle de programmation hybride (MPI+X)*

En mémoire partagée, *performances pas forcément facile à obtenir* :

- OpenMP < 3.0 : création de points de synchronisation (ex : déclenchement simultanés des I/O)
- Placement des données important (first touch)
- Prise en compte du mécanisme de cohérence des caches (false sharing)

Programmation sous forme de tâches

Modèle de programmation sous forme de tâches :

- Programme décrit sous la forme d'un ensemble de tâches à réaliser et de leurs *dépendences*.
- Moteur d'exécutions des tâches : ordonnancement des tâches sur un ensemble de threads (avec des mécanismes avancées, ex : préserver la localité des données, vol de tâches).
- Permet l'expression du plein potentiel parallèle d'un algorithme !

Exemples : OpenMP >= 4.0, Intel oneTBB, StarPU, OmpSs, ...

```
// lambda function for executing an iteration of loop "A"
auto do_loop_b = [] (int i) {
    // lambda function wrapping work(i, j) call [&] captures "i"
    auto do_work = [&] (int j) { work(i, j); };

    // loop "B"
    tbb::parallel_for(0, M, do_work);
};

tbb::parallel_for(0, N, do_loop_b); // loop "A"
```

Accélérateurs

Programmation pour machines hétérogènes :

- Programmation à base de directives toujours possible avec **OpenACC**, **OpenMP** (target) - offloading
- Programmation de noyaux de calculs à destination des devices : **Cuda** (Nvidia), **SYCL** (Intel, pour CPU, GPU Nvidia, GPU AMD, FPGA d'Intel), HIP (AMD pour GPU AMD et Nvidia)
- ... mais aussi programmation OpenCL, Vulcan, Level-Zero
- C, C++ mais aussi Numba, Julia (qui tirent partie de LLVM).

Grosse problématique de **portabilité** :

- API propriétaires (CUDA), implémentation propriétaires (SYCL) ou performances non garanties
- Dépendence, pérennité et support à long terme ? (ex : Cilk)
- Portabilité des performances ?

Portabilité des codes

OpenMP (SIMD, shared-memory, offloading)

Kokkos (Sandia/ORNL, DOE Exascale Computing Project) : bibliothèque de templates C++ qui permet d'utiliser indifféremment CUDA, HIP, SYCL, HPX, OpenMP en *backend*.

- boucles parallèles (parallel_for, parallel_reduce ...)
- interface de tableaux multidimentionnels avec une implémentation adaptée au matériel (ex : boucles imbriquées, first touch sur NUMA)
- bibliothèques de noyaux de calcul
- ...

Portabilité des codes

C = A + B

SYCL

```
#include <iostream>
#include <sycl/sycl.hpp>

int main(int, char**) {
  float h_a[4] { 1.0, 2.0, 3.0, 4.0 };
  float h_b[4] { 4.0, 3.0, 2.0, 1.0 };
  float h_c[4];

  sycl::queue my_queue;
  // create and allocate device objects
  float *d_a = sycl::malloc_device<float>(4, my_queue);
  float *d_b = sycl::malloc_device<float>(4, my_queue);
  float *d_c = sycl::malloc_device<float>(4, my_queue);

  // copy from host to device
  my_queue.memcpy(d_a, h_a, 4*sizeof(float));
  my_queue.memcpy(d_b, h_b, 4*sizeof(float));
  // wait for asynchronous copies to complete
  my_queue.wait();

  my_queue.parallel_for(4, [=](sycl::id<1> idx) {
  // perform vector addition according to own
  // rank number starting at 0
  d_c[idx] = d_a[idx] + d_b[idx];
  });
  my_queue.wait(); // wait for kernel to complete

  // copy results from device to host
  my_queue.memcpy(h_c, d_c, 4*sizeof(float)).wait();

  // free memory
  sycl::free(d_a, my_queue);
  sycl::free(d_b, my_queue);
  sycl::free(d_c, my_queue);

  return 0;
}
```

KOKKOS

```
#include <iostream>
#include <Kokkos_Core.hpp>#include <iostream>

int main(int argc, char** argv) {
  float a[] = { 1.0, 2.0, 3.0, 4.0 };
  float b[] = { 4.0, 3.0, 2.0, 1.0 };

  Kokkos::initialize(argc,argv);
  {
    // wrap host data in View object
    Kokkos::View<float*,Kokkos::HostSpace> h_a(a,4);
    Kokkos::View<float*,Kokkos::HostSpace> h_b(b,4);

    // create device side Views
    Kokkos::View<float*> d_a("a",4);
    Kokkos::View<float*> d_b("b",4);
    Kokkos::View<float*> d_c("c",4);

    // copy from host to device
    Kokkos::deep_copy(d_a,h_a);
    Kokkos::deep_copy(d_b,h_b);

    Kokkos::parallel_for( 4, KOKKOS_LAMBDA (const int& i) {
        d_c(i) = d_a(i) + d_b(i);
      });

    // create host side for output:
    auto h_c = Kokkos::create_mirror_view(d_c);
    Kokkos::deep_copy(h_c,d_c);
  }

  Kokkos::finalize();

  return 0;
}
```

CUDA

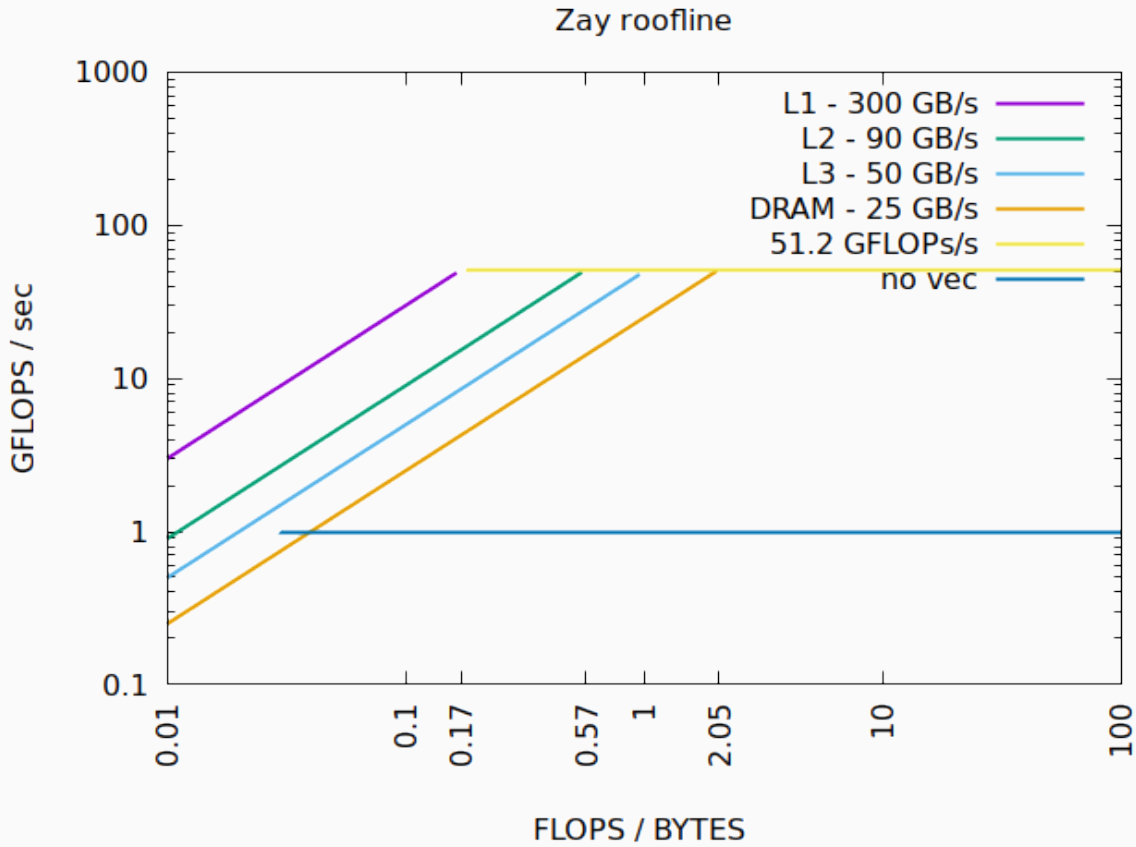
```
#include <iostream>
int main(int argc, char** argv) {

  __global__ void addition(float* a, float* b, float* c){
    int idx = blockIdx.x *blockDim.x + threadIdx.x;
    c[idx] = a[idx] + b[idx];
  }

  int main(int, char**) {
    float h_a[4] { 1.0, 2.0, 3.0, 4.0 };
    float h_b[4] { 4.0, 3.0, 2.0, 1.0 };
    float h_c[4];

    // create and allocate device objects
    float *d_a = 0;
    float *d_b = 0;
    float *d_c = 0;
    cudaMalloc(&d_a,4*sizeof(float));
    cudaMalloc(&d_b,4*sizeof(float));
    cudaMalloc(&d_c,4*sizeof(float));

    cudaMemcpy(d_a,h_a,4*sizeof(float),cudaMemcpyHostToDevice);
    cudaMemcpy(d_b,h_b,4*sizeof(float),cudaMemcpyHostToDevice);

    addition<<<1,4>>>(d_a,d_b,d_c);

    cudaMemcpy(h_c,d_c,4*sizeof(float),cudaMemcpyDeviceToHost);

    // free memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    return 0;
  }
```

Exemple de Taylor Childers, Nathan Nichols (ANL) : Comparing Sycl & Kokkos with MadGraph

La performance

- Critère naturel de performance : FLOP/s
  Ex : $a(i) = b(i) + c(i) * d(i)$ (Fused Multiply-Add)
  5.5 GFLOP/s en L1 SIMD, 2.2 en L1 scalaire, 0.6 en RAM
- Speedup, Efficacité, Strong et Weak scaling...
- **Intensité arithmétique** : nombre de FLOP par octets accédés en mémoire : Ex théorique : $y(i) = a * x(i) + y(i)$
  FLOP = $2n$, bytes = $24n$ (2 loads, 1 store, double = 8 bytes), $IA = 1/12$

La performance



Machine Jean Zay, source D. Lecas, IDRIS

- ERT : Empirical Roofline Toolkit
- Outils debugging, profiling : https://docs.nersc.gov/tools/performance/
- Rester dans le cache (algorithmes par blocs)!

Plan

Comprendre les enjeux

Programmation parallèle

Bibliothèques logiciels

Passage à l'échelle

La parallélisation et l'adaptation des codes à l'architecture est fondamentale mais c'est un travail complexe.

Heureusement, de nombreuses bibliothèques sont disponibles (systèmes linéaires, problèmes aux valeurs propres, optimisation, FFT, EDP, AMR, etc.).

+ : Réduire le temps de développement, faciliter la maintenance ou la lisibilité du code, bénéficier de l'expertise des auteurs

- : Sélection difficile, risque de dépendances à un projet tiers, licences logicielles

Dépend aussi de l'échelle visée (1 ou plusieurs noeuds, GPU, MultiGPU, Exascale ?).

Algèbre linéaire dense en mémoire partagée (CPU ou GPU)

Interfaces standards pour un certains nombres d'opérations :

BLAS : Routines classées en fonction du ratio flops/nb. d'op. mémoire

- Niv. 1 : opérations sur des vecteurs.
$$y = y + \alpha x, \text{ ratio en : } 2n/3n \Rightarrow 2/3 \, (IA = 12)$$
- Niv. 2 : opérations matrices/vecteurs.
$$y = y + Ax, \text{ ratio en } 2n^2/n^2 \Rightarrow 2$$
- Niv. 3 : opérations matrices/matrices.
$$C = C + AB, \text{ ratio en } 2n^3/4n^2 \Rightarrow n/2$$

LAPACK : résolution de systèmes d'équations linéaires, le calcul de valeurs propres et les décompositions de matrices (LU, QR, SVD, Cholesky).

Implémentations libres ou fournies par les constructeurs : MKL (Intel), cuBLAS+cuSOLVER (GPU Nvidia), AOCL (AMD), MAGMA (@utk)
$\Rightarrow$ Utilisez les bibliothèques constructeurs mais attention aux bibliothèques déjà multithreadées! $\Rightarrow$ Privilégier l'écriture des algorithmes avec des BLAS de niveau 3.

Algèbre linéaire dense (distribué)

SLATE est une bibliothèque d'algèbre linéaire dense, distribuée et accélérée pour le GPU, destinée aux systèmes de calcul haute performance (HPC).

- La bibliothèque couvre les fonctionnalitées de BLAS, LAPACK et ScaLAPACK
- Elle comprend des solveurs de systèmes linéaires, des solveurs de moindres carrés, des solveurs de valeurs singulières et de valeurs propres.

img : github.com/icl-utk-edu/slate

Exemple de suites logiciels (multi-noeuds)

Ensembles logiciels intégrés avec des interfaces cohérentes, donnant aussi accès à des outils externes (solveurs directes etc.) :

- PETSc/TAO : *P*ortable, *E*xtensible *T*oolkit for *S*cientific *C*omputation

  www.mcs.anl.gov/petsc/

  - En C “objet’ ’. Très bon support Fortran. Interface Python.
  - Conçu comme une bibliothèque cohérente.
  - Utilisable dans de nombreux packages FEM : FreeFem, Feel++, FEniCS, Firedrake...
- Trilinos : *Building blocks for the development of scientific applications* http://trilinos.org/

  - En C++. Interfaces Fortran 2003 et Python.
  - Organisé comme une collection de *packages*.

Exemple de suites logiciels (Trilinos)

	Objective	Package(s)
Discretizations	Meshing & Discretizations	STKMesh, Intrepid, Pamgen, Sundance, Mesquite
	Time Integration	Rythmos
Methods	Automatic Differentiation	Sacado
	Mortar Methods	Moertel
Services	Linear algebra objects	Epetra, Tpetra
	Interfaces	Xpetra, Thyra, Stratimikos, Piro, …
	Load Balancing	Zoltan, Isorropia, Zoltan2
	“Skins”	PyTrilinos, WebTrilinos, ForTrilinos, Ctrilinos, Optika
	Utilities, I/O, thread API	Teuchos, EpetraExt, Kokkos, Phalanx, Trios, …
Solvers	Iterative linear solvers	AztecOO, Belos, Komplex
	Direct sparse linear solvers	Amesos, Amesos2, ShyLU
	Direct dense linear solvers	Epetra, Teuchos, Pliris
	Iterative eigenvalue solvers	Anasazi
	Incomplete factorizations	AztecOO, Ifpack, Ifpack2
	Multilevel preconditioners	ML, CLAPS, MueLu
	Block preconditioners	Meros, Teko
	Nonlinear solvers	NOX, LOCA
	Optimization	MOOCHO, Aristos, TriKota, Globipack, Optipack
	Stochastic PDEs	Stokhos

Certains paquets utilisent Kokkos.

Matrices et Vecteurs distribués : la description de la distribution parallèle des données utilise une correspondance entre indices locaux et indices globaux et les objets peuvent être redistribués.

Partitionnement de graphes

- Scotch et PT-Scotch http://www.labri.fr/~pelegrin/scotch/
  - Partitionneur séquentiel et parallèle de graphes, de maillages et renumérotation de matrices creuses.
  - Interface de compatibilité METIS.
  - Utilisé notamment par les solveurs MUMPS et PaStiX.
- METIS et ParMETIS https://github.com/KarypisLab/METIS
  - Partitionneur séquentiel et parallèle de graphes, de maillages et renumérotation de matrices creuses.
- Zoltan et Isorropia (Trilinos)

  trilinos.org/capability-areas/meshes-geometry-and-load-balancing/
  - Répartition de charge, partitionnement, coloration de graphes et renumérotation.
  - Méthodes géométriques, graphes et d'hypergraphes.
  - Interfaces pour utiliser les partitionneurs Scotch, PT-Scotch, METIS et ParMETIS.

Valeurs et vecteurs propres

Matrices creuses :

- SLEPc www.grycap.upv.es/slepc/
  - Implémentation parallèle de recherche de valeurs propres.
  - Décomposition en valeurs singulières.
  - Extension de PETSc.
- Anasazi (Trilinos) www.sandia.gov/~rblehou/anasazi-toms.pdf

Entrées/Sorties

- HDF5 (**H**ierarchical **D**ata **F**ormat) : www.hdfgroup.org/HDF5/
  - Format standardisé d'entrées/sorties, séquentielles et parallèles.
  - Existence d'API en C, C++ et Fortran 90.
- NetCDF : www.unidata.ucar.edu/software/netcdf/
  - Bibliothèque et format de données portable.
  - Support du format HDF5 (NetCDF 4.0).
  - Parallel-NetCDF permet d'utiliser le format classique NetCDF en parallèle.
  - Beaucoup utilisé dans les applications de climatologie, de météorologie et d'océanographie.
- MPI-IO :
  - Interface d'entrée-sortie parallèle incluse dans MPI-2.
  - Attention à la portabilité des fichiers binaires.

I/O Systèmes de fichiers distribués : BeeGFS, Lustre, Ceph,... : ils parallélisent les performances de plusieurs supports physiques et noeuds en segmentant les fichiers (blocs distribués sur plusieurs serveurs) => éviter petits fichiers, accès aléatoires ou concurrents sur des petits blocks.

Transformées de Fourier

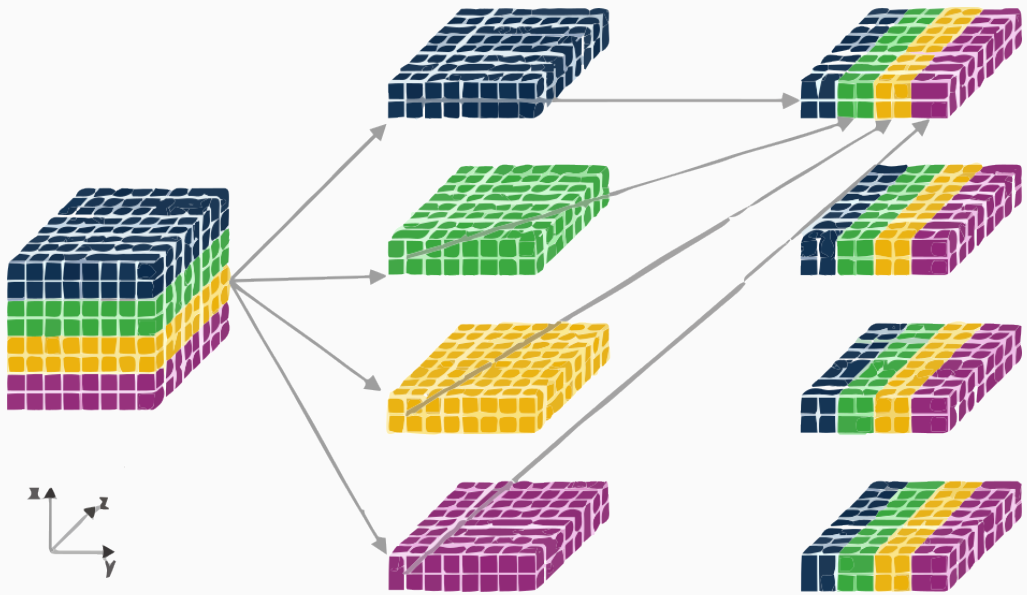
- FFTW (**F**astest **F**ourier **T**ransform in the **W**est) : www.fftw.org
  - Permet d'effectuer des calculs de transformées de Fourier discrètes en une ou plusieurs dimensions.
  - Peut utiliser des threads (pthread ou OpenMP) et MPI.
  - Ecrit en C. Interface Fortran. GPL. Interface de référence

- FFT aussi disponible dans la MKL, cuFFT, cuFFTW ... mais attention à la portabilité.
- Versions Exascale : FFTX, heFFTe

Exécution simultanée par plusieurs processus légers : Penser aux problèmes de **thread safety**. Par exemple, dans FFTW, seule la fonction `fftw_execute` est thread-safe.

Transformées de Fourier

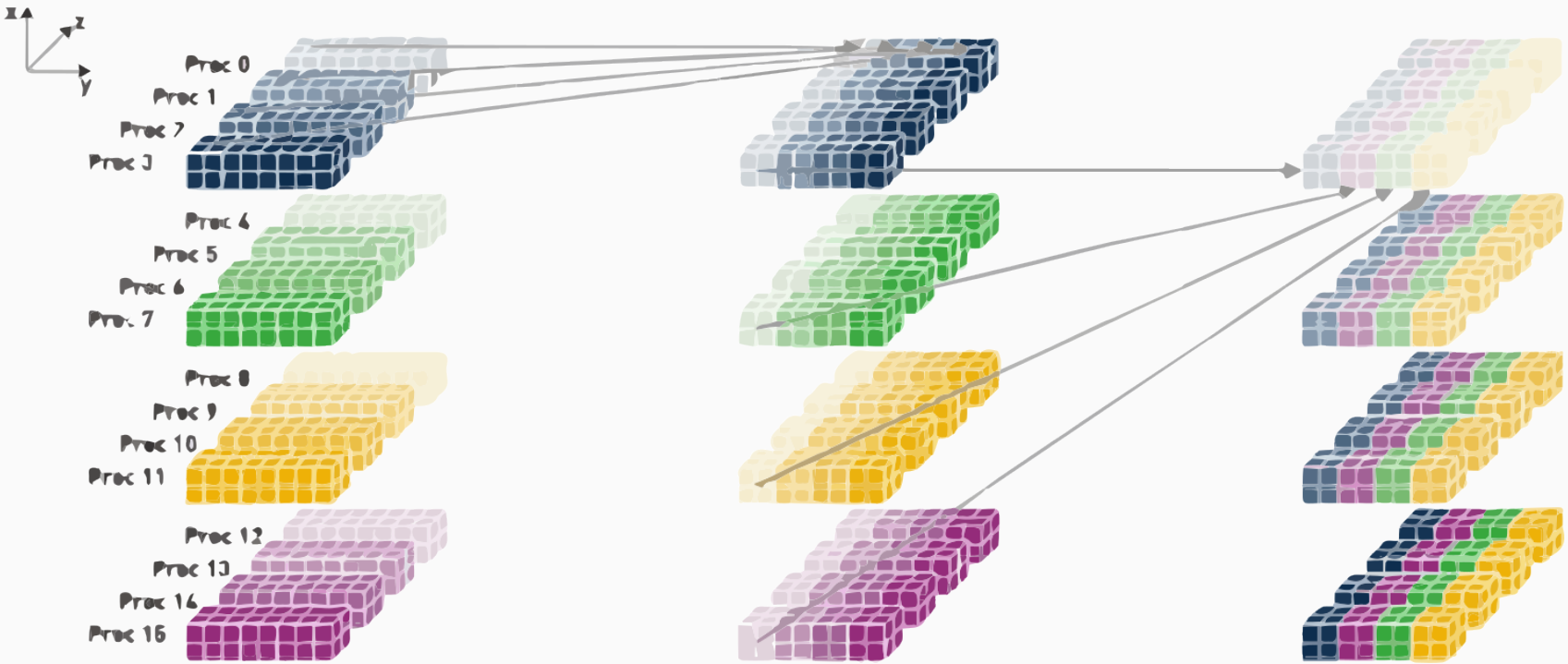
- Slabs (distribution blocs 1D) : FFTW
  - une seule transposée globale pour une transformée 3D. Limite le parallélisme même si threads possibles dans l'autre dimension.
- Pencils (distribution 2D) : P3DFFT, 2DECOMP&FFT, PencilArrays.jl & PencilFFTs.jl



img : Guarrasi, Autotuning of FFTW Library for Massively Parallel Supercomputers

Transformées de Fourier

- Slabs (distribution blocs 1D) : FFTW
  - une seule transposée globale pour une transformée 3D. Limite le parallélisme même si threads possibles dans l'autre dimension.
- Pencils (distribution 2D) : P3DFFT, 2DECOMP&FFT, PencilArrays.jl & PencilFFTs.jl

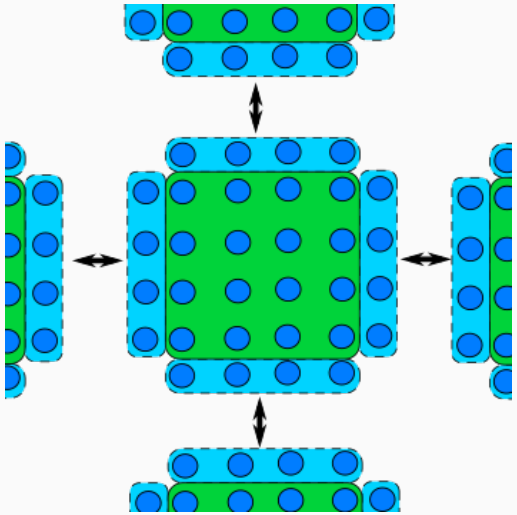


img : Guarrasi, Autotuning of FFTW Library for Massively Parallel Supercomputers

Stencils parallèles

Ex : Méthode de Jacobi, différences finies...

Les bibliothèques de Stencils ont généralement des interfaces de type *Patch* ou *Cellules*.



img : pdc-support.github.io

En Julia : ParallelStencil.jl - Équation de la chaleur 3D MPI+GPUs en 50 lignes ! - écrire un code pour résoudre le problème sur un seul GPU/CPU (grille locale) - trois fonctions suffisent à transformer l'application.

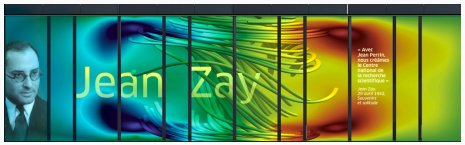
En Python : pystencils + waLBerla

Moyens de calcul nationaux

CINES Adastra

91,6 PFlops, 127k coeurs, 1.5k accélérateurs

IDRIS Jean Zay



125,9 PFlops, 84k coeurs, 3k accélérateurs

TGCC Joliot Curie/Irene

20 Pflops (futur : Jules Vernes)

- GENCI - edari.fr : demandes d'allocations d'heures
- Dossiers *accès dynamique* : au fil de l'eau pour les demandes ≤ 50 kh GPU / 500 kh CPU
- Gratuit pour les chercheurs académiques (et industriels si recherche ouverte)

img : edari.fr

Pour aller plus loin

- Formations IDRIS : MPI, OpenMP, GPU, SIMD, PETSc, Débogage ... (gratuites pour CNRS, Universités)
- Formations PRACE (en ligne) https://events.prace-ri.eu/category/2/
  - Ex : Node-Level Performance Engineering @FAU https://moodle.nhr.fau.de/course/view.php?id=4
- Groupe Calcul, ANF : https://calcul.math.cnrs.fr
- Julia GPU : https://github.com/omlins/julia-gpu-course

Conclusion

- Utiliser au maximum les bibliothèques en s'appuyant sur leurs représentations des données.
- Penser les algorithmes comme un ensemble de tâches et de leurs dépendances.
- L'adaptation des codes aux architectures mémoires est fondamentale.